ocp-memprof: un profileur mémoire pour OCaml

Çağdaş Bozman[1,2,3] & Grégoire Henry[1] & Mohamed Iguernelala[1] & Fabrice Le Fessant[1,2] & Michel Mauny[3]

1: OCamlPro SAS, France,
{cagdas.bozman,gregoire.henry,mohamed.iguernelala}@ocamlpro.com
2: INRIA Paris Rocquencourt, France
fabrice.le_fessant@inria.fr
3: ENSTA-ParisTech, France,
michel.mauny@ensta.fr

Résumé

La récupération automatique de la mémoire est une caractéristique devenue commune des langages de programmation. Elle offre certes au programmeur des garanties de fiabilité, mais, en éloignant ce dernier des détails de la gestion de la mémoire, elle rend plus difficile la compréhension et, *a fortiori*, la maîtrise des allocations de mémoire.

Cet article présente les résultats de nos travaux sur l'analyse du comportement mémoire des programmes OCaml. Nous y expliquerons une technique de profilage basée sur la modification des en-têtes des blocs permettant de récupérer une information de type partielle sur chaque bloc tout en préservant le comportement mémoire du programme original. Ces informations de types sont enrichies ultérieurement par une technique de reconstruction de types par unification. Des algorithmes de graphes peuvent ensuite être appliqués afin d'obtenir des informations plus fines. Nous présentons également quelques cas d'usage montrant comment l'utilisation de ces techniques a permis d'optimiser des logiciels comme Alt-Ergo et Ocsigen.

1. Introduction

La récupération automatique de la mémoire est une caractéristique devenue commune des langages de programmation. Elle offre au programmeur de meilleures garanties de fiabilité : sécurisation contre les pointeurs vers des blocs non-initialisés, ou déjà désalloués, ou encore contre la double libération d'un bloc mémoire, *etc.* Néanmoins, en éloignant le programmeur des détails de la gestion de la mémoire, elle rend plus difficile la compréhension et, *a fortiori*, la maîtrise des allocations.

OCaml, langage multi-paradigme statiquement typé développé à Inria, utilise un ramasse-miettes [8] à deux générations, avec support des pointeurs faibles et de la finalisation des blocs. Grâce à son typage statique fort, OCaml n'a besoin que de très peu d'informations de types sur les valeurs manipulées durant l'exécution de l'application. Par conséquent, la représentation mémoire des valeurs OCaml est plus compacte que pour les autres langages plus faiblement typés, et les performances mémoire sont meilleures.

Dans le cadre de cet article, nous avons étudié d'une façon générale le problème éventuel de comportement mémoire des applications écrites en OCaml, et plus précisément, le cas des fuites mémoires, c'est-à-dire des blocs alloués et non recyclés bien qu'ils soient devenus inutiles au calcul. Dans ce contexte, l'absence presque complète d'information de type dans la représentation des valeurs constitue un obstacle empêchant de fournir au programmeur des informations exploitables sur l'utilisation mémoire de son programme.

Dans une précédente version de notre outil de profilage [4], de l'espace supplémentaire était ajouté, en bytecode, aux blocs mémoire pour être capable de retrouver à l'exécution des informations intéressantes. Cependant, ce changement modifie le comportement des applications profilées, et par la même occasion, celui du ramasse-miettes. Il en résulte deux points négatifs :

— pour les applications allouant beaucoup, le surcoût devient de plus en plus important, le comportement du programme est alors de moins en moins proche de sa version originale. Les informations de déverminage obtenues peuvent alors être de moins en moins précises et justes, vis-à-vis du même programme en production.

— souvent, les programmes profilés sont ceux consommant beaucoup de mémoire. En rajoutant ce surcoût mémoire, les performances de l'application sont nettement dégradées (augmentation du nombre de collections, coût plus important de l'allocation), et l'empreinte mémoire peut augmenter considérablement (manipulation de beaucoup de petits blocs), au point qu'il devient impossible de mettre en production l'application en mode profilage mémoire. Il devient alors très difficile de diagnostiquer un problème ou une fuite mémoire qui ne surviendrait qu'après plusieurs jours ou semaines en production.

Dans cet article, nous proposons une méthode d'instrumentation de la chaîne de compilation et de la bibliothèque d'exécution des programmes qui contourne ces inconvénients. Cette instrumentation est à la base d'outils que nous avons conçus et mis en œuvre, et qui permettent de catégoriser, visualiser et localiser l'utilisation de la mémoire. En particulier, nous avons implanté un nouvel outil, appelé `ocp-memprof` [5], qui sauvegarde la mémoire d'une application OCaml en cours d'exécution, et fournit plusieurs façons d'afficher des informations pertinentes, comme les points d'allocations dans le code source des objets en mémoire et leur type. Il permet également d'agréger les données de plusieurs façons, et ainsi de regrouper les valeurs dans la mémoire par types, modules, fonctions ou localisations dans le code source. Ces modifications ne rajoutent aucun surcoût en mémoire, à l'exception de données allouées temporairement lors de la sauvegarde d'un instantané du tas.

Dans la suite de cet article, nous commencerons par décrire l'architecture et le fonctionnement du profileur. En particulier, nous y détaillerons les différentes modifications apportées à la bibliothèque d'exécution, ainsi qu'au compilateur OCaml. Ensuite, nous présenterons deux applications que nous avons profilées avec `ocp-memprof`. Grâce aux résultats obtenus par notre outil, nous avons pu identifier des causes d'inefficacité dans les deux applications, et proposer des corrections pour ces problèmes.

2. Architecture et fonctionnement du profileur

Afin de pouvoir retrouver le maximum d'informations durant l'exécution, tout en n'engendrant aucun surcoût dans l'occupation du tas, il nous faut pouvoir tracer efficacement les valeurs allouées dans le tas et les lier à un point dans le programme source. Les mécanismes d'allocation étant pour l'essentiel implicites, il n'est pas toujours trivial de tracer le long de la chaîne des compilation le chemin entre un point d'allocation et l'emplacement dans le code source qui demande cette allocation.

L'idée de notre outil est de décorer l'arbre de syntaxe abstraite avec des identifiants uniques pour chaque emplacement qui puisse provoquer une allocation, puis de propager ces identifiants durant la compilation jusqu'aux instructions d'allocation afin de les conserver explicitement dans chaque bloc mémoire. Il se trouve qu'un certain nombre de bits des en-têtes de blocs sont peu utilisés sur les architectures 64 bits. On peut donc s'en servir si on réduit la taille maximale des blocs alloués. C'est dans cet espace que nous stockons les identifiants. Il faut noter qu'en effectuant la génération des identifiants sur l'arbre de syntaxe abtrait typé, on peut associer à chacun d'eux le type *statique* de la valeur correspondante. De plus, durant l'analyse d'un tas mémoire, ces identifiants nous permettent de « remonter » d'un bloc mémoire à son point d'allocation dans le programme source.

2.1. Instrumentation du code

Un identifiant de point d'allocation est un entier dont la valeur maximale peut atteindre $2^{22} - 1$ (soit un peu plus de 4.10^6 possibilités). À la compilation, les points d'allocation de chaque module sont numérotés relativement par rapport à un identifiant de base de ce module. À l'exécution, les identifiants de base des modules sont calculés pour que tous les identifiants des points d'allocation du programme soient consécutifs. Cela permet de mieux exploiter le petit intervalle disponible pour les identifiants. Une erreur est générée à l'exécution si le nombre d'identifiants dépasse les 2^{22} possibilités [1].

Modification des en-têtes La forme d'un bloc mémoire OCaml est rappelée sur la première image de la figure 1. Il est constitué d'un en-tête suivi de différents composants. La partie qui nous intéresse ici est l'en-tête, encodée sur 64 bits. Elle est faite de plusieurs parties, comme le montre la seconde image. Le champ `Wosize`, initialement 54 bits, représente la taille en mots (paquet de 8 octets) d'un bloc. Nous réduisons cette taille à 32 bits, laissant la possibilité de créer des blocs de taille tout-à-fait raisonnable (32 Go), et utilisons les 22 bits ainsi libérés pour y stocker nos identifiants.

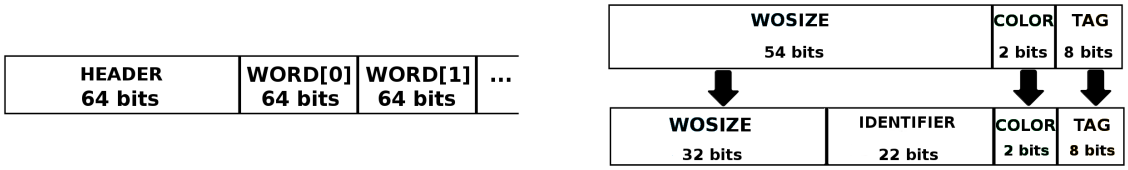

FIGURE 1 – En-tête d'un bloc OCaml & modification pour y ajouter les identifiants d'allocation.

Impact du calcul d'identifiants Les identifiants sont générés sur l'arbre de syntaxe abstraite typé et sont propagés durant les étapes de compilation, que ce soit en bytecode ou en code natif, pour être inclus dans les en-têtes lors de leur génération. Nos expérimentations ont montré que l'addition supplémentaire nécessaire pour placer les identifiants d'allocation dans l'entête des blocs n'avait pas d'impact perceptible sur la vitesse d'exécution des programmes produits.

2.2. Instantanés

L'analyse de la mémoire d'une application OCaml est réalisée sur une image de son graphe mémoire à un moment de son exécution. Nous appelons une telle image un *instantané*. Dans cette section, nous allons voir comment ces instantanés sont créés et ce qu'ils contiennent.

Création d'instantanés Nous avons mis en œuvre trois façons d'obtenir les images de la mémoire, selon le besoin : la première est automatique et est fournie au travers de la variable d'environnement `OCAMLRUNPARAM`. Pour cela, il suffit d'exécuter son programme comme suit :

```
ocp-memprof --exec ./mon-executable.exe
```

`ocp-memprof` se chargera alors de positionner le paramètre `m` de la variable d'environnement `OCAMLRUNPARAM`. Ce paramètre forcera le programme à produire un instantané après chaque collection majeure. Cette production régulière d'instantanés est très utile pour suivre l'évolution de l'utilisation mémoire d'une application donnée durant tout au long de son exécution. Mais cela présente l'inconvénient de produire rapidement des dizaines, voire des centaines de fichiers et, si l'occupation mémoire du processus est conséquente, il est possible de saturer rapidement le disque dur.

1. Cela pourrait être vérifié lors de l'édition de lien du programme, mais l'erreur serait toujours possible avec de la liaison dynamique.

La seconde méthode consiste à produire des instantanés à la demande. Pour cela, il est possible d'envoyer un signal `SIGHUP` au processus qui s'empressera alors de générer une image de la mémoire. Cette méthode peut être très utile en présence d'applications de type serveur dont la durée de vie est infinie, comme MLDonkey ou des serveurs web par exemple.

La troisième et dernière possibilité requiert une annotation légère du code de l'application à profiler pour pouvoir générer des instantanés à des endroits bien précis du code. Pour cela, il suffit d'utiliser la fonction `dump` que nous fournissons dans le module `Heapdump`, et dont la signature est donnée ci-dessous. La chaîne de caractères en entrée permet de nommer les instantanés.

```
external dump: string -> unit = "caml_ml_dumpheap"
```

Contenu des instantanés Les instantanés seront générés dans des fichiers respectant le format de nommage suivant, indépendamment de la méthode de génération utilisée :

`memprof.NUMÉRO-DU-DUMP.NOM-DU-DUMP.dump`

Ces fichiers sont principalement constitués d'une représentation compressée du tas mémoire. Notez que les entiers ne sont pas stockés, et seuls les débuts des chaînes de caractères sont gardés. Pour que la génération des fichiers d'instantané soit la moins coûteuse possible, le tas est parcouru linéairement de la même manière que la phase de nettoyage du ramasse-miette. Ce choix permet de conserver également dans l'instantané, et à moindre coût, l'ensemble des blocs non-alloués et des listes d'allocations.

La figure 2 montre les structures de données définies dans notre bibliothèque d'exécution d'OCaml et utilisées pour la manipulation des images de la mémoire. Les instantanés peuvent être reconstruits en mémoire (type `heap`), parcourus linéairement, ou explorés récursivement depuis les racines (`type roots`). Les fonctions d'exploration, non décrites ici, sont similaires à celles du module `Obj` de la bibliothèque standard d'OCaml (type `value`).

Ces fichiers contiennent aussi quelques informations annexes, comme les informations de statistiques du ramasse-miettes (le type `Gc.stats` usuel), la date de création de l'instantané et une table de correspondance entre nom de symboles et racines globales.

Le type `info` : les différents champs du type `info` sont détaillés ci-dessous :

`hp_pid` : l'identifiant du processus ;

`hp_dump_number` : dans le cas de la production de plusieurs instantanés lors d'une même exécution, ce champ permet de retrouver l'ordre dans lequel les instantanés ont été produits ;

`hp_kind` : indique la nature de l'évènement ayant déclenché la génération de cet instantané : fin d'un GC majeur, signal ou appel explicite à la fonction `dump` (cf. section 2.2) ;

`hp_gc_stat` : informations de statistiques du ramasse-miettes au moment de la génération ;

`hp_dump_start_time / hp_dump_end_time` : date de début et de fin de génération la[2].

Le type `roots` : nous sauvegardons également les racines dans chaque instantané. Le type `roots` est composé des champs suivants :

`hp_globals` : les racines globales

`hp_dyn_globals` : les racines dynamiques qui n'existent pas en bytecode.

`hp_stack` : la pile

`hp_C_globals` : les racines globales en C

`hp_finalised_values` : valeurs finalisées

`hp_hook` : comme par exemple la pile des threads

2. exprimée en *temps processus*.

```
type value
type heap

type kind = Major_gc | Signal | User

type info = {
  hp_pid: int;
  hp_dump_number: int;
  hp_kind: kind;
  hp_gc_stat: Gc.stat;
  hp_dump_start_time: float;
  hp_dump_end_time: float;
}

type roots = {
  hp_globals: (string * value array) array;
  hp_dyn_globals: value array;
  hp_stack: value array;
  hp_C_globals: value array;
  hp_finalised_values: value array;
  hp_hook: value array;
}

type dump = {
  hp_filename: string;
  hp_info: info;
  hp_roots: roots;
  hp_heap: heap;
  hp_globals_map: Heapdump_globals.map;
}
```

FIGURE 2 – Format des instantanés

2.3. Reconstruction de types

En générant les identifiants d'allocation directement sur l'arbre de syntaxe typé, on peut retrouver lors de l'analyse d'un tas mémoire à la fois l'emplacement dans le code source de cette valeur et son type. En pratique, cette information de type est souvent partielle : elle ne correspond qu'au type *statique* de la valeur. Par exemple, dans le cas d'allocation ayant lieu dans la fonction polymorphe telle que `List.map` l'information de type associée au point d'allocation sera α `list`. L'information sur le paramètre de type n'est pas connue statiquement.

La même difficulté apparaît avec les foncteurs : l'information de type associée à un point d'allocation situé à l'intérieur du corps d'un foncteur ne contient pas d'information venant des paramètres de ce foncteur. Cela peut s'avérer gênant avec les foncteurs instanciant des « conteneurs », tel que `Map.Make` par exemple. En effet, une grande partie du tas est constituée de blocs alloués dans `Map.Make` lors de l'analyse de la mémoire, et il n'est pas possible de distinguer simplement si ces blocs proviennent d'une même instanciation du foncteur ou d'instanciations distinctes.

Pour remédier à ce problème, notre outil permet d'effectuer, optionnellement et sur un instantanné donné, une passe de propagation de types. En effet, dans la plupart des situations, les blocs alloués par du code polymorphe sont soit accessibles depuis un bloc alloué dans du code monomorphe, soit contiennent eux-même des blocs alloués dans du code monomorphe. En propageant ces informations de proche en proche par unification, il est possible de reconstruire une grande partie des informations manquantes. Ce travail étant en cours de prototypage, il n'est pas activé par défaut lors du profiling.

2.4. Visualisation des informations

La visualisation des données présentes dans les instantanés nécessite la reconstruction et l'affichage d'informations pertinentes relatives aux graphes mémoire ainsi sauvegardés. L'outil `ocp-memprof` effectue la relecture des instantanés et produit des graphes agrégés de différentes manières.

La figure 3 montre une vue de l'interface graphique d'`ocp-memprof`. Cela consiste en un graphe mémoire et des options permettant différentes manipulations et agrégations sur ce graphe. Cette vue par défaut de l'outil calcule les tailles en mémoire des blocs agrégés par leur type. L'axe des abscisses représente les différents *GC* qui ont eu lieu, et donc les différentes images du tas qui ont été générées. L'axe des ordonnées représente la taille en mots mémoire. L'affichage est quant à lui délégué à la partie javascript de l'outil et est écrit en *js_of_ocaml* [16]. Il utilise la librairie Javascript d3.js [2] pour obtenir différentes vues d'une manière dynamique.

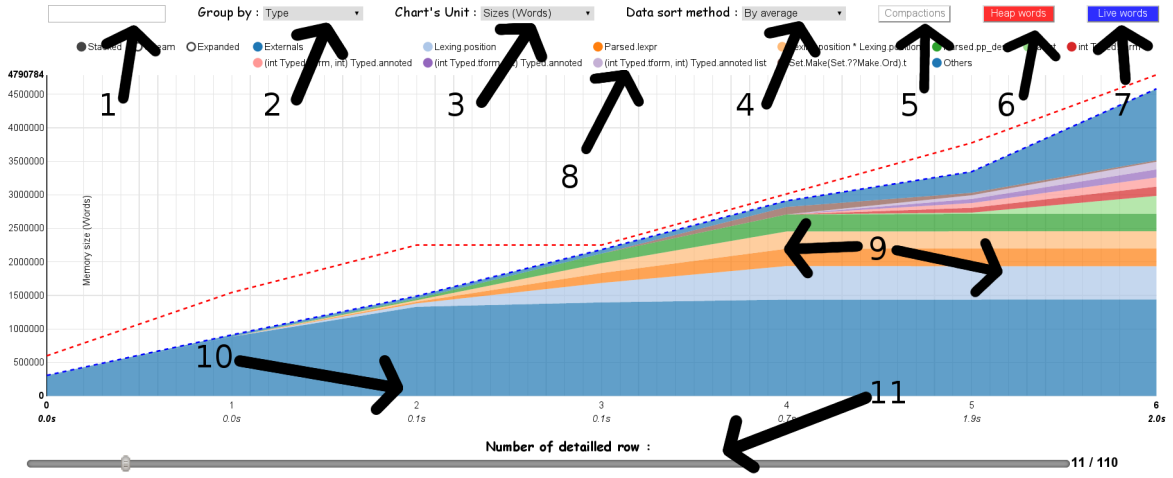

FIGURE 3 – Une vue de l'interface graphique d'`ocp-memprof`

Nous allons maintenant rentrer dans les détails de l'interface graphique. Les différentes parties numérotées sur la figure sont expliquées ci-dessous :

1 Par défaut, seules les 20 premières valeurs sont affichées, par souci de lisibilité. La barre de recherche permet d'ajouter de nouvelles valeurs. Intégrant un module d'auto-complétion, la barre de recherche permet de trouver rapidement ces valeurs non affichées.

2 `ocp-memprof` permet d'agréger les valeurs de différentes manières. Par défaut, les valeurs sont regroupées par leur type.

3 L'outil permet de changer l'unité d'affichage du graphe. Par défaut, celle utilisée par `ocp-memprof` est la taille en mémoire en mots des valeurs. Il est également possible de trier les valeurs par leurs nombres d'occurrences de blocs en mémoire.

4 Il est possible de trier, selon l'unité choisie, les valeurs. Par défaut, `ocp-memprof` trie en faisant la moyenne des valeurs occupant le plus grand espace mémoire sur tous les tas qu'il aura examinés. Il est également possible de trier ces valeurs pour un tas en particulier et d'afficher le résultat sur toute la durée d'exécution du programme.

5 Le bouton *Compactions* permet d'afficher, le cas échéant, les compactions qui ont eu lieu. Cela permet de voir aussi à quel moment le ramasse-miettes a dû procéder à ces compactions.

6 Le bouton *Heap words* permet de dessiner sur le graphe, la courbe de la taille du tas au fil de l'exécution du programme. Comme l'affichage peut ne pas afficher toutes les valeurs présentes en mémoire (*cf.* la limitation par défaut à 20 valeurs), il peut être utile de voir la proportion des valeurs affichées par rapport à la taille totale du tas.

7 De la même manière, le bouton *Live words* permet d'afficher la courbe des valeurs vivantes dans le tas durant l'exécution du programme.
8 Cette zone représente la légende. On peut voir les valeurs, affichées selon l'agrégation choisie, avec leur code couleur
9 Cette zone représente le graphe des valeurs en mémoire, affichées selon les options choisies.
10 L'axe des abscisses représente les différents GC qui ont eu lieu, avec en dessous le temps qui s'est écoulé entre la production de deux instantanés du tas.
11 En plus de la barre de recherche pour ajouter des valeurs au graphe, il y a possibilité de rajouter ou de supprimer des valeurs en utilisant le *slider*.

La figure 4 montre une autre vue du même graphe. Les valeurs sont agrégées par leur identifiant (*locid*) et triées par rapport à l'instantané numéro 42. On voit qu'il y a eu deux compactions aux instantanés 14 et 29. On remarque également que seules les 20 premières valeurs sont affichées (sur 89 disponibles).

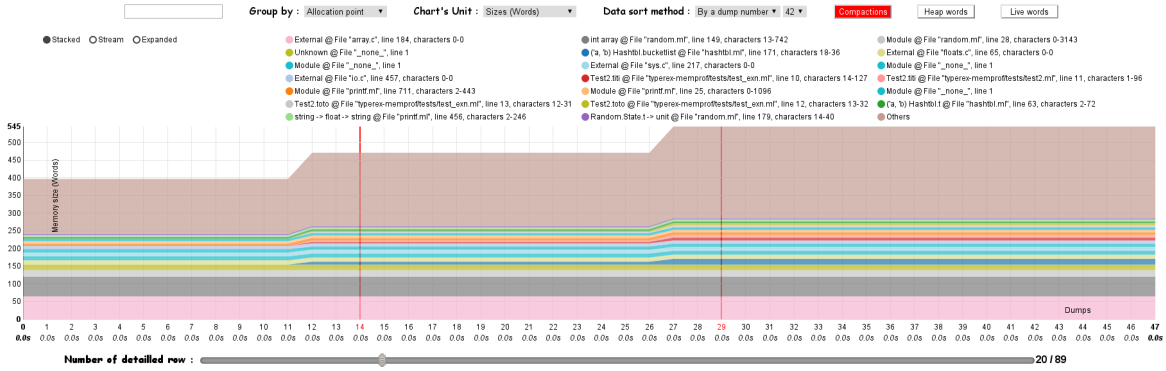

FIGURE 4 – Exemple d'affichage d'un graphe par locid, trié par rapport aux valeurs du 42^{eme} tas.

3. Cas d'utilisation

Dans cette section, nous présentons deux exemples, où l'utilisation d'ocp-memprof a permis de rapidement résoudre des problèmes mémoire pour deux applications réelles, Alt-Ergo et Cumulus.

Ces exemples ont aussi permis de valider certaines hypothèses que nous avions faîtes lors de la conception d'ocp-memprof. Ainsi, le nombre d'identifiants de points d'allocation est de 55 000 pour Alt-Ergo et de 64 000 pour Cumulus, et n'utilisent donc que 16 bits par rapport au 22 bits réservés dans l'en-tête. On observe aussi dans les deux exemples une compression des instantanés qui les ramène à 10 à 20% de la taille mémoire utilisée, et cela bien qu'ils contiennent le graphe complet des pointeurs dans le tas (graphe utilisé, entre autres, pour la reconstruction des types).

3.1. Le démonstrateur automatique Alt-Ergo

Alt-Ergo [1] est un démonstrateur automatique basé sur la technologie SMT (Satisfiability Modulo Theories). Il est utilisé pour montrer la validité de formules logiques issues de la vérification de programmes.

A l'origine se trouve une formule produite par Cubicle [7, 12], lors de la modélisation du protocole de cohérence de cache FLASH, et contenant des conjonctions imbriquées de 999 éléments. L'exécution d'Alt-Ergo sur cette formule provoque une consommation mémoire très importante.

A. Comprendre ce qui se passe en mémoire Nous avons compilé Alt-Ergo avec notre compilateur modifié. Puis, nous avons exécuté `ocp-memprof` sur Alt-Ergo avec la formule générée de la manière suivante :

```
$ ocp-memprof -exec ./alt-ergo.opt -type-only formula.mlw
```

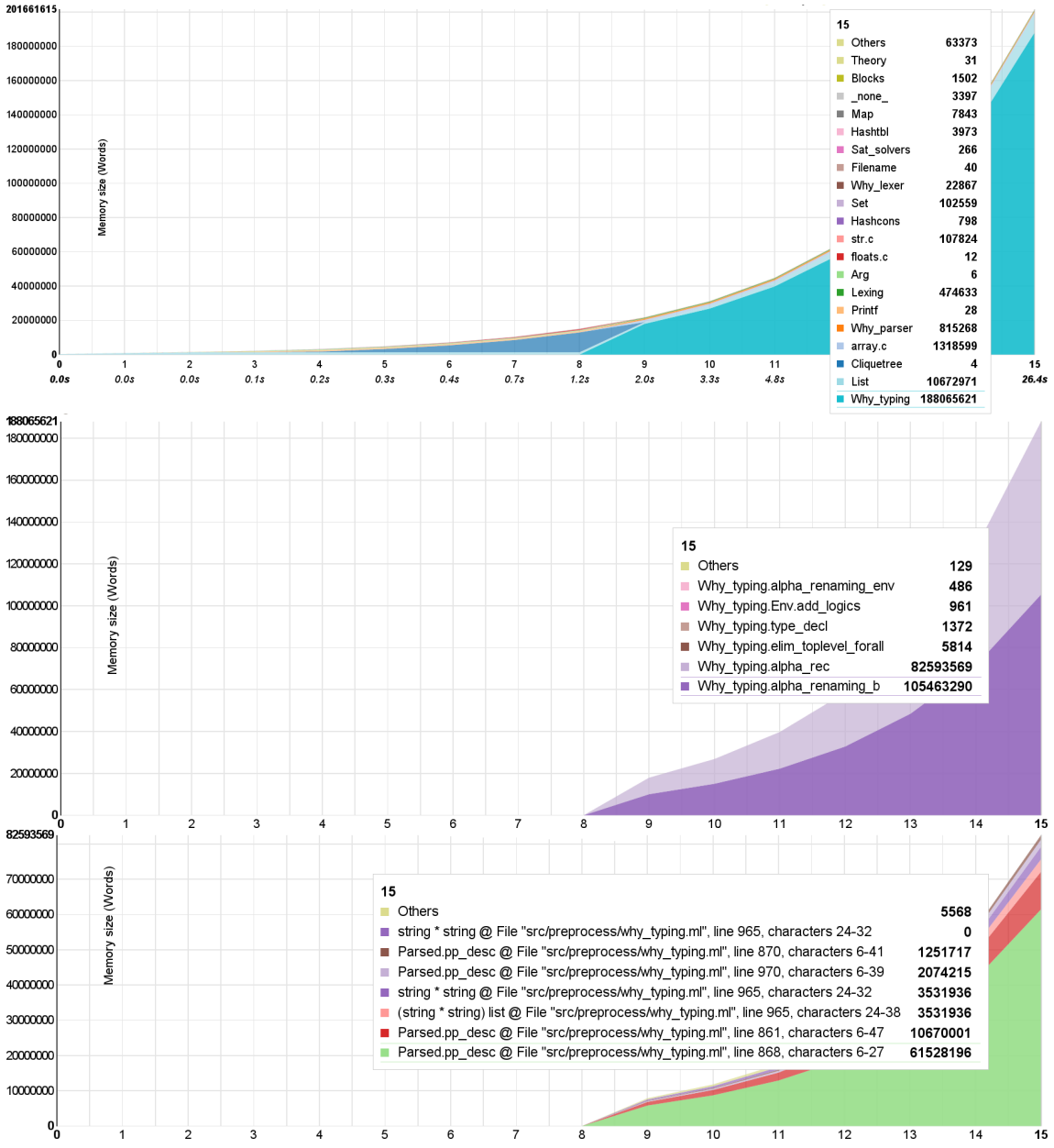

FIGURE 5 – Graphes mémoires générés à l'aide d'`ocp-memprof` sur une exécution d'Alt-Ergo. Le premier graphe est le graphe global trié par modules, le second est obtenu en cliquant sur la courbe Why_typing et le troisième est obtenu en cliquant sur Why_typing.alpha_renaming_b.

Cette commande force la génération d'instantanés de la mémoire après chaque collection majeure du GC. On obtient alors 16 fichiers de dump prêts à être profilés. L'exécution d'Alt-Ergo sur la formule

prend environ 10 secondes sans "le mode de profiling". En activant la génération des instantanés, l'exécution prend 26 secondes.

Après l'analyse des instantanés générés, nous montrons à la figure 5 trois graphes correspondant à trois agrégations différentes. On remarque sur ces graphes que plus de 1.5Go de données sont allouées et jamais libérées. Le premier graphe est une vue dont les valeurs sont agrégées par module. En cliquant sur le module qui prend le plus de place en mémoire, ici `Why_typing`, on obtient le second graphe, représentant le graphe agrégé par les fonctions qui allouent dans ce module. Enfin, en cliquant sur la fonction allouant le plus, c'est-à-dire `alpha_renaming_b`, on obtient le troisième graphe. Ce dernier représente les valeurs agrégées par leur point d'allocation. Les blocs de type `pp_desc`, alloués à la ligne 868 du fichier `src/process/why_typing.ml`, occupent à eux seuls environ 470Mo.

B. Dans le code source, après analyse des graphes L'extrait de code d'Alt-Ergo responsable de cette allocation excessive est montré à la figure 6. La fonction `alpha_renaming_b`, vue dans le deuxième graphe de la figure 5, est une fonction récursive effectuant de l'alpha renommage sur les formules analysées, afin d'éviter la capture des variables. Mais très souvent, il n'y a pas de problème de capture, et la fonction reconstruit une valeur `PPinfix(ff1, op, ff2)` structurellement égale à l'argument `f` de la fonction. Ceci fait que Alt-Ergo alloue énormément sur cette exemple contenant des conjonctions imbriquées de 999 éléments.

```
1  (* Avant de profiler *)
2  let rec alpha_renaming_b s f =
3  ...
4  | PPinfix(f1, op, f2) ->
5    let ff1 = alpha_renaming_b s f1 in
6    let ff2 = alpha_renaming_b s f2 in
7    PPinfix(ff1, op, ff2) (* line 868 *)
8  ...
```

FIGURE 6 – Code de la fonction d'alpha renommage, contenant la ligne 868.

C. Correction du code source L'amélioration du code de la fonction `alpha_renaming_b` est assez facile. Il suffit d'utiliser l'égalité physique pour tester si les appels récursifs ont effectivement renommé dans les arguments ou pas. S'il n'y a pas de renommage dans les arguments, on renvoie simplement l'argument `f` au lieu de reconstruire un objet `PPinfix(ff1, op, ff2)` qui lui est structurellement équivalent. La figure 7 montre l'amélioration de l'extrait de code précédent.

```
1    (* Apres le profileur et le patch *)
2    let rec alpha_renaming_b s f =
3    ...
4    | PPinfix(f1, op, f2) ->
5    let ff1 = alpha_renaming_b s f1 in
6    let ff2 = alpha_renaming_b s f2 in
7    if ff1 == f1 && ff2 == f2 then f
8    else PPinfix(ff1, op, ff2)
9    ...
```

FIGURE 7 – Code de la fonction d'alpha renommage amélioré.

Nous avons ré-exécuté Alt-Ergo sur la même formule après l'amélioration décrite ci-dessus, et relancé `ocp-memprof`. Le nouveau graphe obtenu est montré à la figure 8. Maintenant, Alt-Ergo met 1.8 secondes (2 secondes avec la génération des instantanés) pour typer la même formule et surtout,

il utilise moins de 35Mo de mémoire, diminuant ainsi l'occupation mémoire de façon drastique, et améliorant, par la même occasion, les performances.

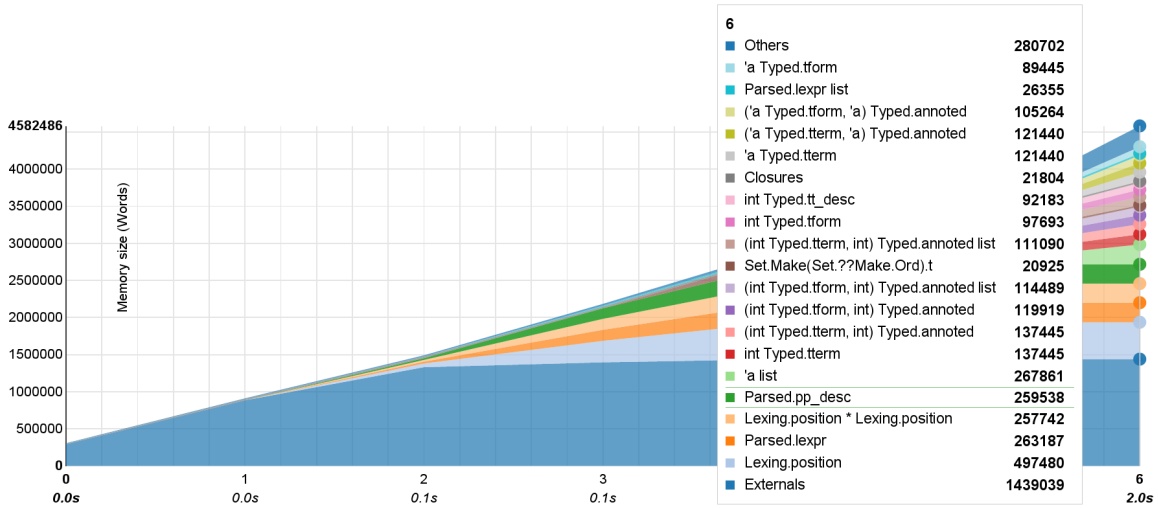

FIGURE 8 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur une exécution d'Alt-Ergo, avec le code patché de la figure 7, sans la reconstruction de types.

3.2. Cumulus, un agrégateur de liens basé sur Ocsigen

Le second cas d'usage que nous présentons est celui d'une fuite mémoire apparaissant dans un serveur HTTP en OCaml. L'agrégateur de liens Cumulus[3] est un site web basé sur le *framework* Eliom du projet Ocsigen. Après avoir compilé Cumulus avec notre compilateur et l'exécuté pendant plusieurs semaines, le processus serveur en production montre une occupation mémoire anormalement élevée. Le programme ayant été initialement compilé avec notre compilateur, nous pouvons lui envoyer le signal `SIGHUP` pour obtenir des instantanés du tas mémoire.

Une fois ces tas rapatriés sur une machine de développement, nous pouvons les analyser avec `ocp-memprof` et obtenir des statistiques sur l'occupation mémoire (figure 9). L'utilisateur averti d'Eliom identifiera rapidement que la majorité de la mémoire est constituée de nœuds et d'attributs d'arbres XML ou HTML, mais aussi des chaînes de caractères et des clôtures.

Malheureusement, il n'est pas immédiat de savoir quels fragments de code de Cumulus sont responsables de l'allocation de ces arbres XML. Ces arbres sont en effet des types abstraits alloués à l'aide de fonctions exportées par des modules d'Eliom, la majorité des points d'allocation est donc située dans le code source d'Eliom.

De manière générale, ce problème de localisation des valeurs ayant un type abstrait est difficile à résoudre simplement avec des statistiques sur les points d'allocation. Il peut être utile de parcourir le graphe mémoire —qui peut être entièrement reconstruit à partir de l'instantané— pour, par exemple, identifier tous les chemins entre les globales et les blocs représentant les nœuds XML.

L'approche que nous adoptons consiste alors à regarder le graphe mémoire afin d'identifier les racines retenant une partie importante de la mémoire. Sur la figure 10, on observe le tableau des tailles retenues, avec le pourcentage, des racines de notre application, sur notre dernier instantané. L'observation qui est faite est que 88,1% de la mémoire est retenue par des finaliseurs. En les regardant

3. `http://cumulus.mirai.fr/`

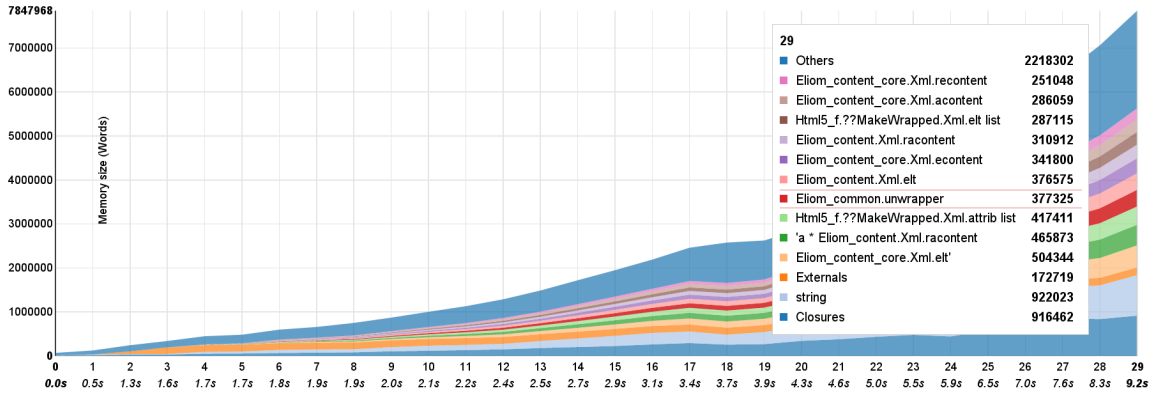

FIGURE 9 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur une utilisation de l'application Cumulus.

de plus près (figure 11), on se rend compte qu'il y a une quantité non négligeable de valeurs de type `Eliom_comet_base.channel_data Lwt_stream.t -> unit` : la fuite mémoire semble donc venir du module `Eliom_comet_base`.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	27159360	100.0%
finalised_values		0	0.0%	23938386	88.1%
stack		0	0.0%	2112583	7.8%
globals		0	0.0%	1040276	3.8%
dyn_globals		0	0.0%	66752	0.2%
hook		0	0.0%	1247	0.0%
c_globals		0	0.0%	116	0.0%

FIGURE 10 – Tableau représentant le graphe mémoire de Cumulus avec la taille retenue par chacune des racines du programme.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	27159360	100.0%
finalised_values		0	0.0%	23938386	88.1%
root221	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit				
root222	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	111730	0.4%
root224	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110579	0.4%
root225	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110579	0.4%
root228	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110577	0.4%
root223	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110571	0.4%
root226	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110571	0.4%

'a Eliom_comet_base.channel_data Lwt_stream.t -> unit @ File "src/lib/eliom_comet.server.ml", line 736, characters 25-58

FIGURE 11 – Tableau représentant le graphe mémoire de Cumulus. Nous avons identifié les racines retenant plus de 88.1% de la mémoire. Chaque nouvelle connexion créée une valeur qui n'est jamais réclamée par le ramasse-miettes et donc les finaliseur associé à ces valeurs ne disparaissent jamais non plus.

La fuite n'étant pas triviale à traquer et corriger, il y a cependant une correction possible dans le cas de Cumulus. En effet, après une recherche plus approfondie dans le code source de l'application, sur la figure 12, la fonction `of_react` peut prendre un argument optionnel `scope` pour préciser quel genre de canal doit être utilisé par la fonction `Eliom_comet.Channel.create`. La valeur `Lwt_stream` associée à ce finaliseur ne semble jamais être récupérée par le ramasse-miettes. Nous modifions alors cette fonction et changeons la valeur par défaut de ce scope par une autre valeur fournie par ce module. Il n'y a alors plus qu'un seul canal qui est créé et tous les clients passent par celui-ci, contrairement à l'autre méthode qui créait un canal par client.

```
1  let (event, call_event) =
2    let (private_event, call_event) = React.E.create () in
3    let event = Eliom_react.Down.of_react
4                  ~scope: Eliom_common.site_scope private_event in
5    (event, call_event)
```

FIGURE 12 – Extrait de code dans Cumulus, avec la fuite mémoire corrigée.

En relançant notre outil d'analyse sur le graphe, avec la méthode de création des canaux modifiée, la valeur retenue par ces racines passent de 88,1% à 0%. Il n'y a plus qu'un canal de communication qui est créé.

On relance alors l'application pour observer la consommation mémoire et on obtient le graphe de la figure 13. L'application passe d'une consommation mémoire de plus de 60Mo pour seulement quelques dizaines de rechargements de la page, à moins de 4Mo pour plusieurs centaines de rechargements de la page ! Mais le plus important à noter ici est que dans la version initiale, la mémoire continuait d'augmenter, symptôme qu'il y avait bien une fuite mémoire. Or, maintenant, la mémoire reste constante comme nous pouvons l'observer sur la figure 13.

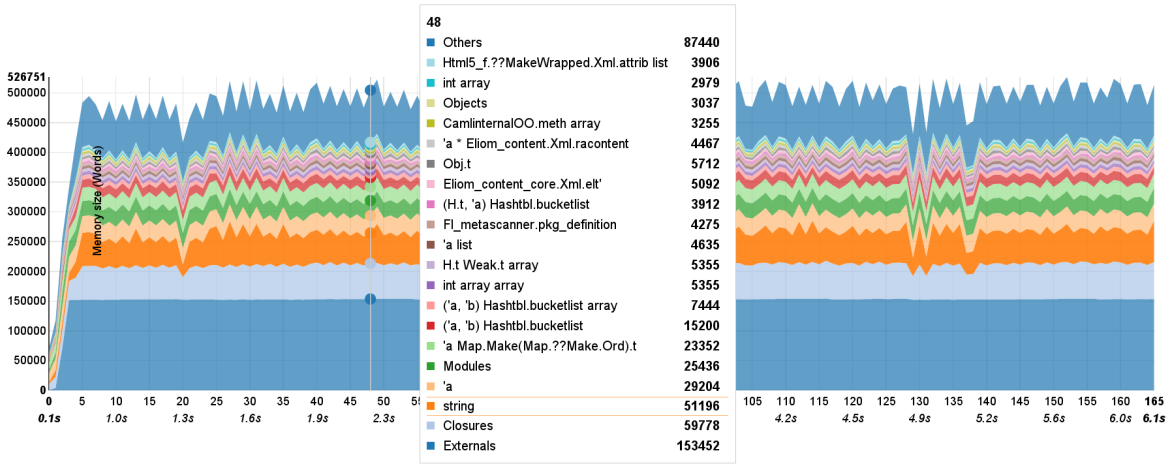

FIGURE 13 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur l'application Cumulus, après réparation de la fuite mémoire. La mémoire reste constante tout au long de l'exécution de l'application.

4. Travaux existants

Dans cette section, nous commençons par présenter les travaux précédents portant uniquement sur OCaml, puis nous élargissons aux travaux effectués sur d'autres langages.

4.1. Profileurs mémoire pour OCaml

Il n'existe, à notre connaissance, que trois travaux sur le profilage mémoire de programmes OCaml :
— ocamlmemprof, qui peut être considéré comme une version précédente de nos travaux ;
— un profileur mémoire basé sur DWARF et développé par Mark Shinwell ;
— et OCamlViz, un outil d'instrumentation de code OCaml.

4.1.1. Le profileur ocamlmemprof

Le profileur ocamlmemprof est un patch du compilateur OCaml 3.07 développé par Fabrice Le Fessant en 2003, conçu à l'origine pour traquer une fuite de mémoire dans le client pair-à-pair MLDonkey [11]. ocamlmemprof a deux modes de fonctionnement, selon qu'il est utilisé en bytecode ou en code natif. Dans les deux cas cependant, des instantanés du tas (*heap snapshots*) sont sauvegardés sur disque, soit à chaque GC majeur, soit par l'appel d'une fonction OCaml, soit à la réception d'un signal Unix particulier. Une première analyse, à partir d'un de ces instantanés, permet de calculer la quantité de mémoire retenue par chaque variable globale.

En bytecode, chaque bloc alloué est étendu avec trois valeurs, indiquant le point d'allocation, le type du bloc et la date d'allocation. L'inconvénient de cette technique est qu'elle perturbe le comportement mémoire du programme de façon importante (un élément de liste, par exemple, voit sa taille doubler en passant de 3 à 6 mots mémoire). Outre un ralentissement de l'exécution, ces modifications peuvent affecter la durée de vie et la promotion des blocs, car les GCs deviennent beaucoup plus fréquents.

En code natif, la plupart des blocs ne sont pas modifiés, mais le compilateur attribue des tags particuliers à chaque enregistrement et à chaque n-uplet. Comme l'espace de ces tags est limité à 240 éléments, de nombreuses collisions sont présentes. Néanmoins, ces tags (et ceux des constructeurs habituels) permettent de retrouver les types des blocs avec une probabilité acceptable.

Le travail présenté ici version comporte trois améliorations notables :
— l'utilisation d'identifiants uniques dans les en-têtes permet de retrouver les points d'allocation et la plupart des types sans perturbation du fonctionnement du GC, ce qui permet d'instrumenter une application en production ;
— la reconstruction des types permet de retrouver les types des blocs dont les identifiants n'ont pas permis d'obtenir une information suffisamment précise ;
— les calculs effectués sur les instantanés permettent de visualiser efficacement les résultats et d'identifier rapidement la source des problèmes mémoire.

4.1.2. Le profileur de Mark Shinwell

En septembre 2013, Mark Shinwell a diffusé un patch [14] du compilateur OCaml 4.01 permettant de profiler une application OCaml, à la fois pour traquer les allocations et observer la durée de vie des blocs. Ce travail s'est inspiré de celui présenté dans cet article pour ajouter à chaque bloc alloué un identifiant dans l'en-tête du bloc, mais cet identifiant est basé sur l'adresse du pointeur de code de l'instruction effectuant l'allocation. Le patch utilise le format DWARF, qui est un format standard de description des informations de déverminage, pour effectuer ensuite une traduction de l'adresse vers la position dans les sources.

Comparé au travail présenté ici, le travail de Shinwell présente plusieurs inconvénients : d'abord, le programme instrumenté doit non seulement contenir des informations de déverminage pour l'ensemble de ses composants, ce qui demande de compiler l'ensemble avec l'option -g, et donc de changer les instructions de compilation, mais il doit aussi faire appel directement à des fonctions de sauvegarde, ce qui demande de savoir exactement où l'observation doit être effectuée. D'autre part, le calcul dynamique de l'en-tête ralentit l'application de façon parfois importante (dans nos tests sur alt-ergo, on observe une augmentation de 70% du temps d'exécution). Enfin, la visualisation des résultats est

minimaliste : des scripts permettent uniquement d'afficher le classement des points d'allocation, sans afficher les types des valeurs correspondantes, qu'il faut donc ensuite aller inspecter dans le code.

4.1.3. L'outil OCamlViz

OCamlViz [6] est un outil d'instrumentation d'applications OCaml, permettant de visualiser facilement en temps réel l'évolution de certains indicateurs du programme. L'application doit être liée statiquement avec une bibliothèque OCaml qui lance un thread de communication en arrière plan. L'application OCamlViz se connecte alors à ce thread, pour recevoir en temps quasi réel les mises à jour des valeurs. Par défaut, seuls les statistiques du GC sont transmises, mais l'utilisateur peut modifier son application pour signaler les modifications de certaines variables. Outre la valeur de la variable, la bibliothèque permet aussi de calculer la quantité de mémoire retenue par la valeur, ou pour des conteneurs (tables de hachage, par exemple), le nombre de valeurs qu'ils stockent.

Comparé au travail présenté ici, OCamlViz ne fournit que des informations très globales sur l'utilisation mémoire, des informations plus fines demandant un lourd travail d'instrumentation du code par l'utilisateur qui ralentirait considérablement les performances de l'application observée. OCamlViz s'adresse donc plus aux utilisateurs voulant observer l'évolution d'autres paramètres de leur application, voire monitorer à distance des compteurs dans une application serveur, sans avoir à construire une interface graphique spécifique.

4.2. Autres travaux

Le langage Java étant doté d'un gestionnaire automatique de mémoire, le problème du profilage des allocations et des fuites de mémoire s'est naturellement posé, et a donné lieu à un certain nombre de travaux et d'outils visant à aider les développeurs d'applications Java.

Parmi les profileurs, on peut citer JProfiler [9] et JProbe [15] qui permettent de profiler et d'analyser la mémoire de programmes Java. Outre le fait qu'ils concernent un langage différent du nôtre, avec notamment un rythme d'allocation très différent, ces outils sont connus pour n'être réellement utilisables qu'en phase de test et non pas en production, à cause du ralentissement notable de la vitesse d'exécution qu'imposent les instrumentations du code qu'ils effectuent.

Il existe aussi des outils cherchant explicitement à détecter les fuites mémoire : c'est le cas d'un des composants de JProbe, qui permet interactivement de comprendre les conséquences de l'élimination de telle ou telle référence, mais aussi d'autres outils comme LeakBot [13] et Cork [10] qui cherchent à identifier les données allouées dont le nombre (et donc la taille globale) augmente de façon considérée comme anormale. Cette technique peut donner lieu à l'émergence de faux-positifs lorsque, par exemple, une augmentation d'allocations a été à tort considérée comme anormale.

Nous devons aussi mentionner l'existence d'outils comme Sleigh [3] dont l'heuristique d'identification de fuites est basée sur le temps écoulé depuis la dernière utilisation d'objets. Là aussi, il y a un risque d'apparition de faux-positifs : par exemple, une *frame* dans Java Swing peut ne jamais être utilisée, sans pour autant devoir être considérée comme une fuite.

5. Conclusion

Dans cet article, nous avons présenté nos travaux sur l'analyse du comportement mémoire des programmes OCaml. En modifiant astucieusement les entêtes des blocs alloués par l'application pour y placer un identifiant du point d'allocation, notre système permet d'obtenir une information très précise sur l'origine et le type de chaque bloc mémoire. Notre système sauvegarde régulièrement des instantanés du tas, qui peuvent ensuite être traités par notre application de visualisation. Celle-ci effectue une propagation des types par unification pour enrichir encore la connaissance des types, et

permet de visualiser de façon efficace l'évolution de l'utilisation mémoire au cours de l'exécution de l'application. Nous avons présenté deux exemples pour lesquels ces techniques ont permis de dévoiler des problèmes mémoire et de les résoudre rapidement.

Nous avons l'intention d'étendre ces travaux dans plusieurs directions : d'une part, nous envisageons d'extraire d'autres informations que celles fournies par les instantanés, par exemple soit en comptant pour chaque point d'allocation les blocs alloués, promus, désalloués ou encore vivants (on obtient alors une vision de toute l'histoire de l'application, non limitée à la seule vision transversale d'un instantané), soit en sauvegardant de façon compressée l'ensemble des allocations/libérations, afin d'être capable de les rejouer dans un simulateur afin d'identifier les paramêtres idéaux du GC ou de potentiels problèmes. D'autre part, nous envisageons d'exécuter des algorithmes de graphes un peu plus sophistiqués sur les instantanés, afin d'exploiter totalement le graphe des pointeurs mémoire que nous avons actuellement à notre disposition. Ces algorithmes pourraient permettre, idéalement de façon automatique, d'identifier des causes de problèmes mémoire, ou d'obtenir des statistiques plus fins que ceux que nous fournissons aujourd'hui (espace retenu par champ d'un enregistrement, *etc.*).

Références

[1] Alt-Ergo, the SMT solver for software verification. `http://alt-ergo.ocamlpro.com/`, `http://alt-ergo.lri.fr/`.

[2] D3.js, a JavaScript library for manipulating documents based on data. `http://d3js.org/`.

[3] M. D. Bond and K. S. McKinley. Bell : bit-encoding online memory leak detection. *SIGPLAN Not.*, 41 :61–72, October 2006.

[4] C. Bozman, M. Mauny, F. Le Fessant, and T. Gazagnaire. Study of OCaml programs' memory behavior. OCAml Users and Developers, 2012.

[5] C. Bozman, M. Mauny, F. Le Fessant, and T. Gazagnaire. Profiling the memory usage of OCaml applications without changing their behavior. OCaml 2013, 2013.

[6] S. Conchon, J.-C. Filliâtre, F. Le Fessant, J. Robert, and G. Von Tokarski. Observation temps-réel de programmes Caml. Vieux-Port La Ciotat, France, 2010. Hermann.

[7] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle : A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper. In Madhusudan and Seshia [12], pages 718–724.

[8] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. 20th symp. Principles of Programming Languages*, pages 113–123. ACM press, 1993.

[9] ej-technologies GmbH. Jprofiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[10] M. Jump and K. S. McKinley. Cork : dynamic memory leak detection for garbage-collected languages. *SIGPLAN Not.*, 42 :31–38, January 2007.

[11] F. Le Fessant and S. Patarin. MLdonkey, a multi-network peer-to-peer file-sharing program. 2003.

[12] P. Madhusudan and S. A. Seshia, editors. *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer, 2012.

[13] N. Mitchell and et al. Leakbot : An automated and lightweight tool for diagnosing memory for diagnosing memory leaks in large applications, 2003.

[14] M. Shinwell. Allocation profiling for x86-64 native code, 2013. `https://github.com/mshinwell/ocaml/tree/4.01-allocation-profiling`.

[15] Q. Software. Jprobe. `http://www.quest.com/jprobe/`.

[16] J. Vouillon and V. Balat. From bytecode to JavaScript : the Js of OCaml compiler.